

# On the Expressive Power of User-Defined Effects

## Effect Handlers, Monadic Reflection, Delimited Control

Yannick Forster<sup>1,2</sup>, Ohad Kammar<sup>2,3</sup>, Sam Lindley<sup>4</sup>, and Matija Pretnar<sup>5</sup>

<sup>1</sup> Saarland University [forster@ps.uni-saarland.de](mailto:forster@ps.uni-saarland.de)

<sup>2</sup> University of Cambridge Computer Laboratory

<sup>3</sup> University of Oxford Department of Computer Science [ohad.kammar@cs.ox.ac.uk](mailto:ohad.kammar@cs.ox.ac.uk)

<sup>4</sup> University of Edinburgh School of Informatics [sam.lindley@ed.ac.uk](mailto:sam.lindley@ed.ac.uk)

<sup>5</sup> University of Ljubljana Faculty of Mathematics and Physics  
[matija.pretnar@fmf.uni-lj.si](mailto:matija.pretnar@fmf.uni-lj.si)

**Abstract.** We compare the expressive power of three programming abstractions for user-defined computational effects: Bauer and Pretnar’s effect handlers, Filinski’s monadic reflection, and delimited control. This comparison allows a precise discussion about the relative merits of each programming abstraction.

We present three calculi, one per abstraction, extending Levy’s call-by-push-value. These comprise syntax, operational semantics, a natural type-and-effect system, and, for handlers and reflection, a set-theoretic denotational semantics. We establish their basic meta-theoretic properties: adequacy, soundness, and strong normalisation. Using Felleisen’s notion of a macro translation, we show that these abstractions can macro-express each other, and show which translations preserve typeability. We use the adequate finitary set-theoretic denotational semantics for the monadic calculus to show that effect handlers cannot be macro-expressed while preserving typeability either by monadic reflection or by delimited control. We supplement our development with a mechanised Abella formalisation.

## 1 Introduction

How should we compare abstractions for user-defined effects?

The use of computational effects, such as file, terminal, and network I/O, random-number generation, and memory allocation and mutation, is controversial in functional programming. While languages like Scheme and ML allow these effects to occur everywhere, pure languages like Haskell restrict the use of effects. The main reason for restricting the use of effects is that computational effects deviate from the lambda calculus, violating its most basic equational properties like  $\beta$ -equality, referential transparency, and confluence. The loss of these properties may lead to unpredictable behaviour in lazy languages like Haskell, or limit the applicability of correctness preserving transformations like common subexpression elimination or code motion.

*Monads* [60] are the established abstraction for incorporating effects into pure languages. The introduction of monads into Haskell led to their additional use

as a programming abstraction, allowing new effects to be declared and used as if they were native. Examples include parsing [27], backtracking and constraint solving [55], and mechanised reasoning [63, 6]. Libraries now exist for monadic programming even in impure languages such as OCaml<sup>6</sup>, Scheme<sup>7</sup>, and C++ [58].

Bauer and Pretnar [5] propose to use *algebraic effects and handlers* to structure programs with user-defined effects. In this approach, the programmer first declares *algebraic operations* as the syntactic constructs she will use to cause the effects, in analogy with declaring new exceptions. Then, she defines *effect handlers* that describe how to handle these operations, in analogy with exception handlers. While exceptions immediately transfer control to the enclosing handler without resumption, a computation may continue in the same position following an effect operation. In order to support resumption, an effect handler has access to the *continuation* at the point of effect invocation. Thus algebraic effects and handlers provide a form of *delimited control*.

Delimited control operators have long been used to encode effects [7] and algorithms with sophisticated control flow [16]. There are many variants of such control operators, and their inter-relationships are subtle [57], and often appear only in folklore.

We study these three different abstractions for user-defined effects: effect handlers, monads, and delimited control operators. Our goal is to enable language designers to conduct a precise and informed discussion about the relative expressiveness of each abstraction. In order to compare them, we build on an idealised calculus for functional-imperative programming, namely call-by-push-value [42], and extend it with each of the three abstractions and their corresponding natural type systems. We then assess the expressive power of each abstraction by rigorously comparing and analysing these calculi.

We use Felleisen’s [14] notion of macro expressibility: when a programming language  $\mathcal{L}$  is extended by some feature, we say that the extended language  $\mathcal{L}_+$  is *macro expressible* when there is a syntax-directed translation from  $\mathcal{L}_+$  to  $\mathcal{L}$  that keeps the features in  $\mathcal{L}$  fixed. Felleisen introduces this notion of reduction to study the expressive power of Turing-complete calculi, as macro expressivity is more sensitive in these contexts than computability and complexity notions of reduction. We adapt Felleisen’s notion to the situation where one extension  $\mathcal{L}_+^1$  of a base calculus  $\mathcal{L}$  is macro expressible in another extension  $\mathcal{L}_+^2$  of the same base calculus  $\mathcal{L}$ . Doing so enable us to formally compare the expressive power for each approach to user-defined effects.

In the first instance, we show that, disregarding types, all three abstractions are macro-expressible in terms of one another, giving six macro-expression translations. Some of these translations are known in less rigorous forms, either published, or in folklore. One translation, macro-expressing effect-handlers in delimited control, improves on previous concrete implementations [30], which rely on the existence of recursive types. The translation from monadic reflection to effect handlers is completely novel.

<sup>6</sup> [http://www.cas.mcmaster.ca/~carette/pa\\_monad/](http://www.cas.mcmaster.ca/~carette/pa_monad/)

<sup>7</sup> <http://okmij.org/ftp/Scheme/monad-in-Scheme.html>

We also establish whether these translations preserve typeability: the translations of some well-typed programs are untypeable. We show that the translation from delimited control to monadic reflection preserves typeability. We conjecture that the converse translation also preserves typeability, but do not yet have a proof. We also give a negative result: we demonstrate how to use the denotational semantics for the monadic calculus to prove that no macro translation exists that preserves typeability. This set-theoretic denotational semantics and its adequacy for Filinski’s multi-monadic metalanguage [20] is another piece of folklore. We conjecture that a similar proof, though with more mathematical sophistication, can be used to prove the non-existence of a typeability preserving macro-expression translation from the monadic calculus to effect handlers. To this end, we give adequate set-theoretic semantics to the effect handler calculus with its type-and-effect system, and highlight the critical semantic invariant a monadic calculus will invalidate. Fig. 1 summarises our contributions and conjectured results. Unlabelled arrows between the typed calculi signify that the corresponding macro translation between the untyped calculi preserves typeability. Arrows labelled by  $\#$  signify that *no* macro translation exists between the calculi, not even a partial macro translation that is only defined for well-typed programs.

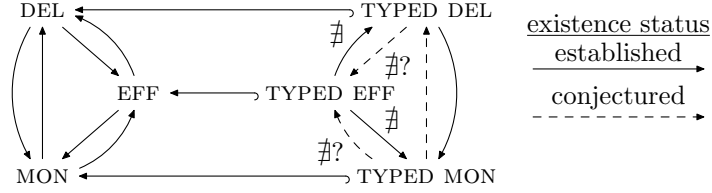


Fig. 1: Existing and conjectured macro translations

We supplement our pencil-and-paper proofs with a mechanised formalisation in the Abella proof assistant [22, 23], which complement, rather than duplicate, our formal development. Specifically, we prove a Felleisen-style [62] progress-and-preservation soundness theorem for each calculus, and only present three of our translations in prose, leaving the folklore to our formalisation.

We make the following contributions:

- three formal calculi, i.e., syntax and semantics, for effect handlers, monadic reflection, and delimited control extending a shared call-by-push-value core, and their meta-theory:
  - adequate set-theoretic denotational semantics for effect handlers;
  - adequate set-theoretic denotational semantics for monadic reflection;
  - a denotational soundness proof for effect handlers and delimited control;
  - strong normalisation for monadic reflection and delimited control;
- six macro-translations between the three untyped calculi;

- formally mechanised meta-theory in Abella<sup>8</sup> comprising:
  - progress and preservation theorems;
  - the translations between the untyped calculi; and
  - their correctness proofs in terms of formal simulation results;
- typeability preservation of the macro translation from delimited control to monadic reflection; and
- that there exists no typeability-preserving macro translation from effect handlers to either monadic reflection or delimited control.

We structure the remainder of the paper as follows. Sections 2, 3, 4, and 5 present the core calculus and its extensions with effect handlers, monadic reflection, and delimited control, respectively, and their meta-theoretic properties. Section 6 presents the macro translations between these calculi, their correctness, and typeability preservation. Section 7 concludes and outlines further work.

## 2 The core-calculus: MAM

We are interested in a functional-imperative calculus where effects and higher-order features interact well. Levy’s [42] call-by-push-value (CPBV) calculus serves this purpose. The CPBV paradigm subsumes call-by-name and call-by-value, both syntactically and semantically. In CPBV evaluation order is explicit, and the way it combines computational effects with higher-order features yields simpler program logic reasoning principals [48, 29]. CPBV allows us to uniformly deal with call-by-value and call-by-name evaluation strategies, making the theoretical development relevant to both ML-like and Haskell-like languages. We extend it with a type-and-effect system, and, as *adjunctions* form the semantic basis for CPBV, we call the resulting calculus the *multi-adjunctive metalanguage* (MAM).

|                             |  |                                    |
|-----------------------------|--|------------------------------------|
| $V, W ::=$ values           | $M, N ::=$ computations                            | $V!$ force                         |
| $x$ variable                | <b>case</b> $V$ <b>of</b> product                  | <b>return</b> $V$ returner         |
| $()$ unit value             | $(x_1, x_2) \rightarrow M$ matching                | $x \leftarrow M; N$ sequencing     |
| $(V_1, V_2)$ pairing        | <b>case</b> $V$ <b>of</b> { variant                | $\lambda x.M$ abstraction          |
| $\text{inj}_\ell V$ variant | $\text{inj}_{\ell_1} x_1 \rightarrow M_1$ matching | $M V$ application                  |
| $\{M\}$ thunk               | $:$  | $\langle M_1, M_2 \rangle$ pairing |
|                             | $\text{inj}_{\ell_n} x_n \rightarrow M_n$          | $\text{prj}_i M$ projection        |

Fig. 2: MAM syntax

Fig. 2 presents MAM’s raw term syntax, which distinguishes between values (data) and computations (processes). We assume a countable set of variables ranged over by  $x, y, \dots$ , and a countable set of variant constructor literals ranged over by  $\ell$ . The unit value, product of values, and finite variants/sums

<sup>8</sup> <https://github.com/matijapretnar/user-defined-effects-formalization>

are standard. A computation can be suspended as a thunk  $\{M\}$ , which may be passed around. Products and variants are eliminated with standard pattern matching constructs. Thunks can be forced to resume their execution. A computation may simply return a value, and two computations can be sequenced, as in Haskell's `do` notation. Function computations abstract over a value to which they can be applied. In order to pass a function as data, it must first be thunked. For completeness, we also include CBPV's binary computation products, which subsume projections on product values in call-by-name languages.

$$\begin{array}{ll}
\textbf{Frames and contexts } \mathcal{B} ::= x \leftarrow []; N \mid [] V \mid \mathbf{prj}_i [] & \text{basic frames} \\
\mathcal{F} ::= \mathcal{B} & \text{computation frames} \\
\mathcal{C} ::= [] \mid \mathcal{C}[\mathcal{F}[]] & \text{evaluation context} \\
\mathcal{H} ::= [] \mid \mathcal{H}[\mathcal{B}[]] & \text{hoisting context} \\
\\
\textbf{Beta reduction } \boxed{M \rightsquigarrow_\beta M'} & \\
(\times) \text{ case } (V_1, V_2) \text{ of } (x_1, x_2) \rightarrow M \rightsquigarrow_\beta M[V_1/x_1, V_2/x_2] & \left( (U) \quad \{M\}^! \rightsquigarrow_\beta M \right. \\
(+) \text{ case } \mathbf{inj}_\ell V \text{ of } \{ \dots \mathbf{inj}_\ell x \rightarrow M \dots \} \rightsquigarrow_\beta M[V/x] & \left( (\rightarrow) \quad (\lambda x. M) V \rightsquigarrow_\beta M[V/x] \right. \\
(F) \quad x \leftarrow \mathbf{return } V; M \rightsquigarrow_\beta M[V/x] & \left. (\&) \quad \mathbf{prj}_i \langle M_1, M_2 \rangle \rightsquigarrow_\beta M_i \right) \\
\\
\textbf{Reduction } \boxed{M \rightsquigarrow M'} & \frac{M \rightsquigarrow_\beta M'}{\mathcal{C}[M] \rightsquigarrow \mathcal{C}[M']}
\end{array}$$

Fig. 3: MAM operational semantics

Fig. 3 presents MAM's standard structural operational semantics, in the style of Felleisen [15]. In order to reuse the core definitions as much as possible, we refactor the semantics into  $\beta$ -reduction rules and a single congruence rule. As usual, a  $\beta$ -reduction reduces a matching pair of introduction and elimination forms. We specify in the definition of evaluation contexts the *basic frames*, which all our extensions will share. Later, in each calculus we will make use of *hoisting frames* in order to capture continuations, stacks of basic frames, extending from a control operator to the nearest delimiter. As usual, a reducible term can be decomposed into at most one unique pair of evaluation context and  $\beta$ -reducible term, making the semantics deterministic.

In this development, we use the following standard syntactic sugar. We use nested patterns in our pattern matching constructs. We allow the application of functions and the elimination constructs to arbitrary computations, and not just values, by setting for example  $M \ N := x \leftarrow N; M \ x$  for some fresh  $x$ , giving a more readable, albeit call-by-value, appearance.

Fig. 4 presents MAM's types and effects. It is a variant of Kammar and Plotkin's [29] multi-adjunctive intermediate language without effect operations or coercions. As a core calculus for three calculi with very different notions of effect, MAM is pure, and the only shared effect is the empty effect  $\emptyset$ . We include a kind system, unneeded in traditional CBPV where a context-free distinction between values and computations forces types to be well-formed. The two points of difference from CBPV are the kind of effects, and the refinement of the compu-

tation kind by well-kinded effects  $E$ . The other available kinds are the standard value kind and a kind for well-formed environments (without type dependencies). Our type system includes value type variables (which we will later use for defining monads parametrically). Simple value types are standard CBPV value types, and each type of thunks includes an effect annotation describing the effects of these thunks. Computation types include returners  $FA$ , which are computations that return a value of type  $A$ , similar to the monadic type **Monad**  $m \Rightarrow m\ a$  in Haskell. Functions are computations and only take values as arguments. We include CBPV's computation products, which account for product elimination via projection in call-by-name languages. To ensure the well-kindedness of types, which may contain type-variables, we use type environments in a list notation that denotes sets of type-variables. Similarly, we use a list notation for environments, which are functions from a finite set of variable names to the set of *value* types.

|                     |              |  |               |  |                   |
|---------------------|--------------|--|---------------|--|-------------------|
| $E ::=$             | effects      | $A, B ::=$                                 | value types   | $C, D ::=$                                       | computation types |
| $\emptyset$         | pure effect  | $\alpha$                                   | type variable | $FA$   | returners         |
| $K ::=$             | kinds        | $  1$                                      | unit          | $  A \rightarrow C$                              | functions         |
| $  \mathbf{Eff}$    | effects      | $  A_1 \times A_2$                         | products      | $  C_1 \ \& \ C_2$                               | products          |
| $  \mathbf{Val}$    | values       | $  \{\mathbf{inj}_{\ell_1} A_1$            | variants      |  | environments:     |
| $  \mathbf{Comp}_E$ | computations | $  \dots \mid \mathbf{inj}_{\ell_n} A_n\}$ |               | $\Theta ::= \alpha_1, \dots, \alpha_n$           |                   |
| $  \mathbf{Ctx}$    | environments | $  U_E C$                                  | thunks        | $\Gamma, \Delta ::= x_1 : A_1, \dots, x_n : A_n$ |                   |

Fig. 4: MAM kinds and types

Fig. 5 presents the kind and type systems. The only effect ( $\emptyset$ ) is well-kinded. Type variables must appear in the current type environment, and they are always value types. The remaining value and computation types and environments have straightforward structural kinding conditions. Thunks of  $E$ -computations of type  $C$  require the type  $C$  to be well-kinded, which includes the side-condition that  $E$  is a well-kinded effect. This kind system has the property that each valid kinding judgement has a unique derivation. Value type judgements assert that a value term has a well-formed value type under a well-formed environment in some type variable environment. The rules for simple types are straightforward, and note how the effect annotation moves between the  $E$ -computation type judgement and the type of  $E$ -thunks. The side condition for computation type judgements asserts that a computation term has a well-formed  $E$ -computation type under a well-formed environment for some well-formed effect  $E$  under some type variable environment. The rules for variables, value and computation products, variants, and functions are straightforward. The rules for thunking and forcing ensure the computation's effect annotation agrees with the effect annotation of the thunk. The rule for **return** allows us to return a value at any effect annotation, reflecting the fact that this is a *may*-effect system: the effect annotations track which effects may be caused, rather than a more prescriptive description. The rule for sequencing reflects our choice to omit any form of effect coercion, subeffecting,

or effect polymorphism: the three effect annotations must agree. There are more sophisticated effect system which allow more flexibility [32]. We conjecture such mechanisms do not change the expressivity of each abstraction qualitatively.

For uniformity's sake, we let types  $X$  range over both value and  $E$ -computation types, and phrases  $P$  range over both value and computation terms. Judgements of the form  $\Theta; \Gamma \vdash_E P : X$  are meta-judgements, ranging over value judgements  $\Theta; \Gamma \vdash P : X$  and  $E$ -computation judgement  $\Theta; \Gamma \vdash_E P : X$ .

For the purpose of contextual equivalence, define the subclass of *ground types*:

$$(\text{ground values}) \quad G ::= 1 \mid G_1 \times G_2 \mid \{\mathbf{inj}_{\ell_1} G_1 \mid \dots \mid \mathbf{inj}_{\ell_n} G_n\}$$

The type of booleans **bit** is  $\{\mathbf{inj}_{\text{False}} 1 \mid \mathbf{inj}_{\text{True}} 1\}$ . The definition of program contexts  $\mathcal{X}[\ ]$  and their type judgements is straightforward but tedious and lengthy with four kinds of judgements, and we omit it. Type judgements

$$\Theta; \Gamma \vdash_E \mathcal{X}[\ ] : X[\Theta'; \Delta \vdash_{E'} Y : ]$$

assert that the hole in the program context expects a term of type  $Y$ , and possible effect  $E'$ , well-typed in type variable environment  $\Theta'$  and environment  $\Delta$ . The context will then be typeable in type variable environment  $\Theta$  and environment  $\Gamma$ , and will be a well-typed term of type  $X$ , and possible effect  $E$ . Finally, we say that an environment  $\Gamma'$  *extends* an environment  $\Gamma$ , and write  $\Gamma' \geq \Gamma$  if  $\Gamma'$  extends  $\Gamma$  as a partial function from identifiers to value types.

Let  $\Theta; \Gamma \vdash_E P, Q : X$  be two MAM phrases. We say that  $P$  and  $Q$  are *contextually equivalent* and write  $\Theta; E \vdash_\Gamma P \simeq Q : X$  when, for all *closed* well-typed *ground-returner* contexts

$$; \vdash_\emptyset \mathcal{X}[\ ] : FG[\Theta'; \Gamma' \vdash_E X : ]$$

with  $\Theta' \supseteq \Theta$ ,  $\Gamma' \geq \Gamma$  and for all closed ground value terms  $; \vdash V : G$ , we have:

$$\mathcal{X}[P] \rightsquigarrow^* \mathbf{return} V \iff \mathcal{X}[Q] \rightsquigarrow^* \mathbf{return} V$$

Our operational semantics is sufficiently well-behaved that for all well-typed computations  $\Theta; \Gamma \vdash_E M, M' : C$ , if  $M \rightsquigarrow M'$  then  $M \simeq M'$ . This property will hold for each of our calculi.

MAM has a straightforward set-theoretic denotational semantics. Presenting the semantics for the core calculus will simplify our later presentation. To do so, we first recall the following established facts about monads, specialised and concretised to the set-theoretic setting.

A monad is a triple  $\langle T, \mathbf{return}, \gg \rangle$  where  $T$  assigns to each set  $X$  a set  $TX$ ,  $\mathbf{return}$  assigns to each set  $X$  a function  $\mathbf{return}^X : X \rightarrow TX$  and  $\gg$  assigns to each function  $f : X \rightarrow TY$  a function  $\gg f : TX \rightarrow TY$ , and moreover these assignments satisfy well-known algebraic identities. Given a monad  $\langle T, \mathbf{return}, \gg \rangle$  we define for every function  $f : X \rightarrow Y$  the functorial action  $\mathbf{fmap} f : TX \rightarrow TY$  as  $\mathbf{fmap} f \, xs := xs \gg (\mathbf{return} \circ f)$ . A *T-algebra* for a monad  $\langle T, \mathbf{return}, \gg \rangle$  is a pair  $C = \langle |C|, c \rangle$  where  $|C|$  is a set and  $c : T|C| \rightarrow |C|$

$$\begin{array}{l}
\text{Effect kinding} \quad \boxed{\Theta \vdash_k E : \mathbf{Eff}} \quad \overline{\Theta \vdash_k \emptyset : \mathbf{Eff}} \\
\text{Value kinding} \quad \boxed{\Theta \vdash_k A : \mathbf{Val}} \\
\\
\frac{\alpha \in \Theta}{\Theta \vdash_k \alpha : \mathbf{Val}} \quad \frac{}{\Theta \vdash_k 1 : \mathbf{Val}} \quad \frac{\Theta \vdash_k A_1 : \mathbf{Val} \quad \Theta \vdash_k A_1 : \mathbf{Val}}{\Theta \vdash_k A_1 \times A_2 : \mathbf{Val}} \\
\frac{\text{for every } 1 \leq i \leq n: \Theta \vdash_k A_i : \mathbf{Val}}{\Theta \vdash_k \{\mathbf{inj}_{\ell_1} A_1 \mid \dots \mid \mathbf{inj}_{\ell_n} A_n\} : \mathbf{Val}} \quad \frac{\Theta \vdash_k C : \mathbf{Comp}_E}{\Theta \vdash_k U_E C : \mathbf{Val}} \\
\\
\text{Computation kinding} \quad \boxed{\Theta \vdash_k C : \mathbf{Comp}_E} \quad (\Theta \vdash_k E : \mathbf{Eff}) \\
\\
\frac{\Theta \vdash_k A : \mathbf{Val}}{\Theta \vdash_k FA : \mathbf{Comp}_E} \quad \frac{\Theta \vdash_k A : \mathbf{Val} \quad \Theta \vdash_k C : \mathbf{Comp}_E}{\Theta \vdash_k A \rightarrow C : \mathbf{Comp}_E} \\
\frac{\Theta \vdash_k C_1 : \mathbf{Comp}_E \quad \Theta \vdash_k C_2 : \mathbf{Comp}_E}{\Theta \vdash_k C_1 \& C_2 : \mathbf{Comp}_E} \\
\\
\text{Context kinding} \quad \boxed{\Theta \vdash_k \Gamma : \mathbf{Ctxt}} \quad \frac{\text{for all } x \in \text{Dom}(\Gamma): \Theta \vdash_k \Gamma(x) : \mathbf{Val}}{\Theta \vdash_k \Gamma : \mathbf{Ctxt}} \\
\\
\text{Value typing} \quad \boxed{\Theta; \Gamma \vdash V : A} \quad (\Theta \vdash_k \Gamma : \mathbf{Ctxt}, A : \mathbf{Val}) \\
\\
\frac{(x : A) \in \Gamma}{\Theta; \Gamma \vdash x : A} \quad \frac{}{\Theta; \Gamma \vdash () : 1} \quad \frac{\Theta; \Gamma \vdash V_1 : A_1 \quad \Theta; \Gamma \vdash V_2 : A_2}{\Theta; \Gamma \vdash (V_1, V_2) : A_1 \times A_2} \\
\frac{\Theta; \Gamma \vdash V : A_i}{\Theta; \Gamma \vdash \mathbf{inj}_{\ell_i} V : \{\mathbf{inj}_{\ell_1} A_1 \mid \dots \mid \mathbf{inj}_{\ell_n} A_n\}} \quad \frac{\Theta; \Gamma \vdash_E M : C}{\Theta; \Gamma \vdash \{M\} : U_E C} \\
\\
\text{Computation typing} \quad \boxed{\Theta; \Gamma \vdash_E M : C} \quad (\Theta \vdash_k \Gamma : \mathbf{Ctxt}, E : \mathbf{Eff}, C : \mathbf{Comp}_E) \\
\\
\frac{\Theta; \Gamma \vdash V : A_1 \times A_2 \quad \Theta; \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Theta; \Gamma \vdash_E \mathbf{case } V \mathbf{ of } (x_1, x_2) \rightarrow M : C} \quad \frac{\Theta; \Gamma \vdash V : U_E C}{\Theta; \Gamma \vdash_E V! : C} \\
\frac{\Theta; \Gamma \vdash V : \{\mathbf{inj}_{\ell_1} A_1 \mid \dots \mid \mathbf{inj}_{\ell_n} A_n\} \quad \text{for every } 1 \leq i \leq n: \Theta; \Gamma, x_i : A_i \vdash_E M_i : C}{\Theta; \Gamma \vdash_E \mathbf{case } V \mathbf{ of } \{\mathbf{inj}_{\ell_1} x_1 \rightarrow M_1; \dots; \mathbf{inj}_{\ell_n} x_n \rightarrow M_n\} : C} \\
\frac{\Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathbf{return } V : FA} \quad \frac{\Theta; \Gamma \vdash_E M : C_1 \& C_2}{\Theta; \Gamma \vdash_E \mathbf{prj}_i M : C_i} \\
\frac{\Theta; \Gamma \vdash_E M : FA \quad \Theta; \Gamma, x : A \vdash_E N : C}{\Theta; \Gamma \vdash_E x \leftarrow M; N : C} \quad \frac{\Theta; \Gamma, x : A \vdash_E M : C}{\Theta; \Gamma \vdash_E \lambda x. M : A \rightarrow C} \\
\frac{\Theta; \Gamma \vdash_E M : A \rightarrow C \quad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E M V : C} \quad \frac{\Theta; \Gamma \vdash_E M_1 : C_1 \quad \Theta; \Gamma \vdash_E M_2 : C_2}{\Theta; \Gamma \vdash_E \langle M_1, M_2 \rangle : C_1 \& C_2}
\end{array}$$

Fig. 5: MAM kind and type system



is a function satisfying  $c(\mathbf{return} \ x) = x$ , and  $c(\mathbf{fmap} \ c \ xs) = c(xs \gg id)$  for all  $x \in |C|$  and  $xs \in T^2 |C|$ . The set  $|C|$  is called the *carrier* and we call  $c$  the *algebra structure*. For each set  $X$ , the pair  $FX := \langle TX, \gg id \rangle$  forms a  $T$ -algebra called the *free  $T$ -algebra over  $X$* .

$$\begin{array}{ll}
\textbf{Effects} & \llbracket \emptyset \rrbracket_\theta := \langle \text{Id}, \text{id}, \lambda f.f \rangle \\
\textbf{Values} & \llbracket \alpha \rrbracket_\theta := \theta(\alpha) \quad \llbracket 1 \rrbracket_\theta := \{\star\} \quad \llbracket A_1 \times A_2 \rrbracket_\theta := \llbracket A_1 \rrbracket_\theta \times \llbracket A_2 \rrbracket_\theta \quad \llbracket U_E C \rrbracket_\theta := |\llbracket C \rrbracket_\theta| \\
& \llbracket \{\text{inj}_{\ell_1} A_1 \mid \dots \mid \text{inj}_{\ell_n} A_n\} \rrbracket_\theta := (\{\ell_1\} \times \llbracket A_1 \rrbracket_\theta) \cup \dots \cup (\{\ell_n\} \times \llbracket A_n \rrbracket_\theta) \\
\textbf{Computations} & \\
& \llbracket FA \rrbracket_\theta := F\llbracket A \rrbracket_\theta \quad \llbracket A \rightarrow C \rrbracket_\theta := \langle |\llbracket C \rrbracket_\theta|^{\llbracket A \rrbracket_\theta}, \lambda f_s. \lambda x. c(\mathbf{fmap} \ (\lambda f.f(x)) \ f_s) \rangle \\
& \llbracket C_1 \ \& \ C_2 \rrbracket_\theta := \langle |\llbracket C_1 \rrbracket_\theta| \times |\llbracket C_2 \rrbracket_\theta|, \lambda c_s. \langle c_1(\mathbf{fmap} \ \pi_1 \ c_s), c_2(\mathbf{fmap} \ \pi_2 \ c_s) \rangle \rangle
\end{array}$$

Fig. 6: MAM denotational semantics for types

We parameterise MAM's semantics function  $\llbracket \Theta \vdash_k E : \mathbf{Eff} \rrbracket$  by an assignment  $\theta$  of sets  $\theta(\alpha)$  to each of the type variables  $\alpha$  in  $\Theta$ . Given such a type variable assignment  $\theta$ , we assign to each

- effect: a monad  $\llbracket \Theta \vdash_k E : \mathbf{Eff} \rrbracket_\theta$ , denoted by  $\langle T_{\llbracket E \rrbracket_\theta}, \mathbf{return}^{\llbracket E \rrbracket_\theta}, \gg^{\llbracket E \rrbracket_\theta} \rangle$ ;
- value type: a set  $\llbracket \Theta \vdash_k A : \mathbf{Val} \rrbracket_\theta$ ;
- $E$ -computation type: a  $T_{\llbracket E \rrbracket_\theta}$ -algebra  $\llbracket \Theta \vdash_k C : \mathbf{Comp}_E \rrbracket_\theta$ ; and
- context: the set  $\llbracket \Theta \vdash_k \Gamma : \mathbf{Ctx} \rrbracket_\theta := \prod_{x \in \text{Dom}(\Gamma)} \llbracket \Gamma(x) \rrbracket_\theta$ .

Fig. 6 defines the standard set-theoretic semantics function over the structure of types. The pure effect denotes the identity monad, which sends each set to itself, and extends a function by doing nothing. The extended languages in the following sections will assign more sophisticated monads to other effects. The semantics of type variables uses the type assignment given as parameter. The unit type always denotes the singleton set. Product types and variants denote the corresponding set-theoretic operations of cartesian product and disjoint union, and thus the empty variant type  $0 := \{\}$  denotes the empty set. The type of thunked  $E$ -computations of type  $C$  denotes the carrier of the  $T_{\llbracket E \rrbracket_\theta}$ -algebra  $\llbracket C \rrbracket_\theta$ . The  $E$ -computation type of  $A$  returners denotes the free  $\llbracket E \rrbracket_\theta$ -algebra. Function and product types denote well-known algebra structures over the sets of functions and pairs, correspondingly [3, Theorem 4.2, e.g.].

Terms can have multiple types, for example the function  $\lambda x. \mathbf{return} \ x$  has the types  $1 \rightarrow 1$  and  $0 \rightarrow 0$ , and type judgements can have multiple type derivations. We thus give a Church-style semantics [54] by defining the semantics function for type judgement derivations rather than for terms. To increase readability, we write  $\llbracket P \rrbracket$  instead of including the entire typing derivation for  $P$ .

The semantics function for terms is parameterised by an assignment  $\theta$  of sets to type variables. It assigns to each well-typed derivation for a:

- value term: a function  $\llbracket \Theta; \Gamma \vdash V : A \rrbracket_\theta : \llbracket \Gamma \rrbracket_\theta \rightarrow \llbracket A \rrbracket_\theta$ ; and

- $E$ -computation term: a function  $\llbracket \Theta; \Gamma \vdash_E M : C \rrbracket_\theta : \llbracket \Gamma \rrbracket_\theta \rightarrow |\llbracket C \rrbracket_\theta|$ .

Fig. 7 defines the standard set-theoretic semantics function over the structure of derivations. The semantics of sequencing uses the Kleisli extension function  $(\gg f) : TX \rightarrow |\llbracket C \rrbracket|$  for functions into non-free algebras  $f : X \rightarrow |\llbracket C \rrbracket|$ , given by  $(\gg f) := c \circ \mathbf{return} \circ f$ .

$$\begin{aligned} \text{Value terms} \quad \llbracket x \rrbracket_\theta(\gamma) &:= \pi_x(\gamma) & \llbracket () \rrbracket_\theta(\gamma) &:= \star & \llbracket \mathbf{inj}_\ell V \rrbracket_\theta(\gamma) &:= \langle \ell, \llbracket V \rrbracket_\theta(\gamma) \rangle \\ \llbracket (V_1, V_2) \rrbracket_\theta(\gamma) &:= \langle \llbracket V_1 \rrbracket_\theta(\gamma), \llbracket V_2 \rrbracket_\theta(\gamma) \rangle & \llbracket \{M\} \rrbracket_\theta(\gamma) &:= \llbracket M \rrbracket_\theta(\gamma) \end{aligned}$$

**Computation terms**

$$\begin{aligned} \llbracket \mathbf{case} V \mathbf{ of } (x_1, x_2) \rightarrow M \rrbracket_\theta(\gamma) &:= \llbracket M \rrbracket_\theta(\gamma[x_1 \mapsto a_1, x_2 \mapsto a_2]), \text{ where } \llbracket V \rrbracket_\theta(\gamma) = \langle a_1, a_2 \rangle \\ \llbracket \mathbf{case} V \mathbf{ of } \{ \mathbf{inj}_{\ell_1} x_1 \rightarrow M_1, \dots, \mathbf{inj}_{\ell_n} x_n \rightarrow M_n \} \rrbracket_\theta &:= \begin{cases} \llbracket M_1 \rrbracket_\theta(\gamma[x_1 \mapsto a_1]) & \llbracket V \rrbracket_\theta(\gamma) = \langle \ell_1, a_1 \rangle \\ \vdots \\ \llbracket M_n \rrbracket_\theta(\gamma[x_n \mapsto a_n]) & \llbracket V \rrbracket_\theta(\gamma) = \langle \ell_n, a_n \rangle \end{cases} \\ \llbracket \langle \rangle \rrbracket_\theta(\gamma) &:= \star & \llbracket V! \rrbracket_\theta(\gamma) &:= \llbracket V \rrbracket_\theta(\gamma) & \llbracket \mathbf{return} V \rrbracket_\theta(\gamma) &:= \mathbf{return} (\llbracket V \rrbracket_\theta(\gamma)) \\ \llbracket x \leftarrow M; N \rrbracket_\theta(\gamma) &:= \llbracket M \rrbracket_\theta(\gamma) \gg \lambda a. \llbracket N \rrbracket_\theta(\gamma[x \mapsto a]) \\ \llbracket \lambda x. M \rrbracket_\theta(\gamma) &:= \lambda a. \llbracket M \rrbracket_\theta(\gamma[x \mapsto a]) & \llbracket M V \rrbracket_\theta(\gamma) &:= (\llbracket M \rrbracket_\theta(\gamma))(\llbracket V \rrbracket_\theta(\gamma)) \\ \llbracket \langle M_1, M_2 \rangle \rrbracket_\theta(\gamma) &:= \langle \llbracket M_1 \rrbracket_\theta(\gamma), \llbracket M_2 \rrbracket_\theta(\gamma) \rangle & \llbracket \mathbf{prj}_i M \rrbracket_\theta(\gamma) &:= \pi_i(\llbracket M \rrbracket_\theta(\gamma)) \end{aligned}$$

Fig. 7: MAM denotational semantics for terms

We prove adequacy using standard logical relations techniques, i.e., by defining a relational interpretation to types and establishing a basic lemma. We use the lifting in Hermida’s [25] thesis to define the monadic lifting of a relation.

**Theorem 1 (adequacy).** *Denotational equivalence implies contextual equivalence: for all  $\Theta; \Gamma \vdash_E P, Q : X$ , if  $\llbracket P \rrbracket = \llbracket Q \rrbracket$  then  $P \simeq Q$ .*

We strengthen the existing CBPV’s strong normalisation theorem [11, 12]:

**Corollary 2 (soundness and strong normalisation).** *All well-typed closed ground returners must reduce to a unique normal form: for all  $; \vdash_\emptyset M : FG$  there exists some  $; \vdash V : G$  such that  $\llbracket \mathbf{return} V \rrbracket = \llbracket M \rrbracket$  and  $M \rightsquigarrow^* \mathbf{return} V$ .*

Our Abella formalisation further contains progress and preservation theorems.

In the following sections, we will extend the MAM calculus using the following convention. We use an ellipsis to mean that a new definition consists of the old definition verbatim with the new description appended, as in the following:

$$M, N ::= \dots \mid \mathbf{op} V \quad \text{effect operation}$$

### 3 Effect handlers: EFF

Bauer and Pretnar [5] propose algebraic effects and handlers as a basis for modular programming with user-defined effects. Programmable effect handlers arose

as part of Plotkin and Power’s [49] algebraic account of computational effects, which investigates the consequences of using the additional structure in algebraic presentations of monadic models of effects. This account refines Moggi’s [47] monadic account by incorporating into the theory the syntactic constructs that generate effects as *algebraic operations for a monad* [50]: each monad is accompanied by a collection of syntactic operations, whose interaction is specified by a collection of equations, i.e., an algebraic theory, which fully determines the monad. To fit exception handlers into this account, Plotkin and Pretnar [51] generalise to the handling of arbitrary algebraic effects, giving a computational interpretation to algebras for a monad. By allowing the user to declare operations, the user can describe new effects in a composable manner. By defining algebras for the free monad with these operations, users give the abstract operations different meanings similarly to Swierstra’s [59]’s use of free monads.

|                                      |                    |   |                  |
|--------------------------------------|--------------------|---|------------------|
| $M, N ::= \dots$                     | computations       | $H ::=$                                     | handlers         |
| $  \text{op } V$                     | operation call     | $\{\text{return } x \mapsto M\}$            | return clause    |
| $  \text{handle } M \text{ with } H$ | handling construct | $  H \uplus \{\text{op } p \ k \mapsto N\}$ | operation clause |

(a) Syntax extensions to Fig. 2

|  |   |   |
|--|---|---|
| <b>Frames and contexts</b>                             | $\dots \mathcal{F} ::= \dots   \text{handle } [ \ ] \text{ with } H$  | computation frame   |
| <b>Beta reduction</b>                                  | $\dots$   | For every $H = \{\text{return } x \mapsto N_{\text{ret}}\} \uplus \{\text{op}_1 \ p_1 \ k_1 \mapsto N_1\} \uplus \dots$ |
| $(\text{ret}) \text{handle } (\text{return } V)$       | $\text{with } H \rightsquigarrow_\beta M[V/x]$  | $\uplus \{\text{op}_n \ p_n \ k_n \mapsto N_n\}$  |
| $(\text{op}) \text{handle } \mathcal{H}[\text{op } V]$ | $\text{with } H \rightsquigarrow_\beta N[V/p, \{\lambda x. \text{handle } \mathcal{H}[\text{return } x] \text{ with } H\}/k]$ |   |

(b) Operational semantics extensions to Fig. 3

Fig. 8: EFF

Fig. 8(a) presents the extension EFF, Kammar et al.’s [30] core calculus of effect handlers. We assume a countable set of elements of a separate syntactic class, ranged over by  $\text{op}$ . We call these *operation names*. For each operation name  $\text{op}$ , EFF’s operation call construct allows the programmer to cause the effect associated with  $\text{op}$  by passing it a value as an argument. Operation names are the only interface to effects the language has. The handling construct allows the programmer to use a handler to interpret the operation calls in a given returner computation. As the given computation may call thunks returned by functions, the decision which handler will handle a given operation call is dynamic. Handlers are specified by two kinds of clauses. A *return clause* describes how to proceed when we return a value. A *operation clause* describes how to proceed when we an operation  $\text{op}$ . The body of an operation clause can access the value passed in the operation call using the first bound variable  $p$ , which is similar to the bounding occurrence of an exception variable when handling exceptions. But unlike exceptions, we expect arbitrary effects like reading from or writing to memory to resume. Therefore the body of an operation clause can also access the continuation at the operation’s calling point. Even though we use a list notation

in this presentation of the syntax, the abstract syntax tree representation of a handler  $H$  is in fact a pair  $H = \langle H^{\text{return}}, H^- \rangle$  consisting of a single return clause  $H^{\text{return}}$ , and a function  $H^-$  from a finite subset of the operation names assigning to each operation name  $op$  its associated operation clause  $H^{op}$ .

Fig. 8(b) presents EFF's extension to MAM's operational semantics. Computation frames  $\mathcal{F}$  now include the handling construct, while the basic frames  $\mathcal{B}$  do not, allowing a handled computation to  $\beta$ -reduce under the handler. We add two  $\beta$ -reduction cases. When the returner computation inside a handler is fully evaluated, the return clause proceeds with the return value. When the returner computation inside a handler needs to evaluate an operation call, the definition of hoisting contexts  $\mathcal{H}$  ensures  $\mathcal{H}$  is precisely the continuation of the operation call delimited by the handler. Put differently, it ensures that the handler in the root of the reduct is the closest handler to the operation call in the call stack. The operation clause corresponding to the operation called then proceeds with the supplied parameter and current continuation. Rewrapping the handler around this continuation ensures that all operation calls invoked in the continuation are handled in the same way. The alternative [30, 35, 43] is to define instead:

$$\text{handle } \mathcal{H}[op \ V] \text{ with } H \rightsquigarrow_{\beta} N[V/p, \{\lambda x. \mathcal{H}[\text{return } x]\}/k]$$

This variant is known as *shallow* handlers, as opposed to the *deep* handlers of Fig. 8(b). We focus on deep handlers that are closer to monadic reflection. See Pretnar's [53] tutorial for additional exposition to programming with handlers.

|                                       |                  |                                  |                       |
|---------------------------------------|------------------|----------------------------------|-----------------------|
| $E ::= \dots$                         | effects          | $K ::= \dots$                    | kinds                 |
| $  \{op : A \rightarrow B\} \uplus E$ | arity assignment | $  \mathbf{Hndlr}$               | handlers              |
|                                       |                  | $R ::= A \xRightarrow{E}^{E'} C$ | handler types $\dots$ |

Fig. 9: Kinds and types extension to Fig. 4

Fig. 9 presents EFF's types and effects. The effect annotations in EFF are functions from finite sets of operation names, assigning to each operation name its parameter type  $A$  and its return type  $B$ . We add a new kind for handler types, which describe the kind and the returner type the handler can handle, and the kind and computation type the handling clause will have.

Fig. 10 presents how EFF extends MAM's kind system. The types in each operation's arity assignment must be value types. The kinding judgement for handlers requires all the types and effects involved to be well-kinded. Computation type judgements now include two additional rules for each of the new computation constructs. An operation call is well-typed when the parameter and return type agree with the arity assignment in the effect annotation. A use of the handling construct is well-typed when the type and effect of the handled computation and the type-and-effect of the construct agree with the types and effects in the handler type. The set of operations the handler can handle must strictly agree with the set of operations in the effect annotation in the type. The variable

$$\begin{array}{l}
\textbf{Effect kinding} \quad \dots \quad \frac{\Theta \vdash_k A : \mathbf{Val} \quad \Theta \vdash_k B : \mathbf{Val} \quad \text{op} \notin E \quad \Theta \vdash_k E : \mathbf{Eff}}{\Theta \vdash_k \{\text{op} : A \rightarrow B\} \uplus E : \mathbf{Eff}} \\
\textbf{Handler kinding} \quad \boxed{\Theta \vdash_k R : \mathbf{Hndlr}} \quad \frac{\Theta \vdash_k A : \mathbf{Val} \quad \Theta \vdash_k E, E' : \mathbf{Eff} \quad \Theta \vdash_k C : \mathbf{Comp}_{E'}}{\Theta \vdash_k A \xRightarrow{E \Rightarrow E'} C : \mathbf{Hndlr}} \\
\textbf{Computation typing} \\
\dots \quad \frac{(\text{op} : A \rightarrow B) \in E \quad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \text{op } V : FB} \quad \frac{\Theta; \Gamma \vdash_E M : FA \quad \Theta; \Gamma \vdash H : A \xRightarrow{E \Rightarrow E'} C}{\Theta; \Gamma \vdash_{E'} \text{handle } M \text{ with } H : C} \\
\textbf{Handler typing} \quad \boxed{\Theta; \Gamma \vdash H : R} \quad (\Theta \vdash_k \Gamma : \mathbf{Ctx}, R : \mathbf{Hndlr}) \\
\frac{\Theta; \Gamma, x : A \vdash_E M : C \quad \text{for all } 1 \leq i \leq n: \quad \Theta; \Gamma, p : A_i, k : U_E(B_i \rightarrow C) \vdash_E N_i : C}{\Theta; \Gamma \vdash \{\mathbf{return } x \mapsto M\} \uplus \{\text{op}_i \ p \ k \mapsto N_i \mid 1 \leq i \leq n\} : A^{\{\text{op}_i : A_i \rightarrow B_i \mid 1 \leq i \leq n\}} \xRightarrow{E} C}
\end{array}$$

Fig. 10: Kinding and typing extensions to Fig. 5

bound to the return value has the returner type in the handler type. In each operation clause, the bound parameter variable has the parameter type from the arity assignment for this operation, and the continuation variable's input type matches the return type in the operation's arity assignment. The overall type of all operation clauses agrees with the computation type of the handler. The second effect annotation on the handler type matches the effect annotations on the continuation and the body of the operation and return clauses, in accordance with the deep handler semantics.

EFF's ground types are the same as MAM's. We omit the full definition of program contexts, and define contextual equivalence as in MAM.

EFF's design involves several decisions. First, handlers have their own kind, unlike Pretnar's [53] calculus in which they are values. This distinction is minor, as handlers as values can be expressed by thunking the handling construct. Next, the effect annotations involved in the handling construct have to agree precisely. The other option is to check inclusion of operation sets, i.e., a handler may handle more effects than the annotation on the effect. This distinction is minor, as we can express coercions from an effect annotation into a superset of effects using a trivial handler:

$$\{\lambda x. \mathbf{return } x\} \uplus \{\text{op } p \ k \mapsto k(\text{op } p) \mid \text{op} \in E\} : A \xRightarrow{E \Rightarrow E'} FA$$

A more significant choice is to use *closed* handlers: execution halts/crashes when a handled computation calls an operation the handler does not handle. The other option is to use *forwarding* handlers [30], in which unhandled operation calls are forwarded to the nearest enclosing handler that can handle them. In our simple type-and-effect system, this decision has no immediate impact, as we can use the trivial handler above to re-raise unhandled effects whenever needed. However, in more expressive type systems, which we do not consider here, in particular type systems with *effect polymorphism* [44, 41, 26], this distinction is more significant.

In this case, we believe that the language should include both variants: the forwarding variant to support code extensibility and modularity, and the closed variant to allow the programmer to guarantee that a computation cannot cause unhandled effects. Finally, it is possible to remove the effect system. In that case, the arity assignments for the operations need to be placed globally at the top level of the program, as in Pretnar’s [53] tutorial. Removing the effect system has dramatic consequences on expressivity: as we are about to see, well-typed EFF terms are strongly normalising. If we remove the effect annotations, we can encode a form of Landin’s [40] knot, making the calculus non-terminating.

We give an adequate set-theoretic denotational semantics for EFF. First, recall the following well established concepts in universal and categorical algebra. A *signature*  $\Sigma$  is a pair consisting of a set  $|\Sigma|$  whose elements we call *operation symbols*, and a function  $\text{arity}_\Sigma$  from  $|\Sigma|$  assigning to each operation symbol  $f \in |\Sigma|$  a (possibly infinite) set  $\text{arity}(f)$ . We write  $(f : A) \in \Sigma$  when  $f \in |\Sigma|$  and  $\text{arity}_\Sigma(f) = A$ . Given a signature  $\Sigma$  and a set  $X$ , we inductively form the set  $T_\Sigma X$  of  $\Sigma$ -terms over  $X$  by:

$$t ::= x \mid f \langle t_a \rangle_{a \in A} \quad (x \in X, (f : A) \in \Sigma)$$

The assignment  $T_\Sigma$  together with the following assignments form a monad

$$\mathbf{return} \ x := x \quad t \gg f := t[f(x)/x]_{x \in X} \quad (f : X \rightarrow T_\Sigma Y)$$

The  $T_\Sigma$ -algebras  $\langle C, c \rangle$  are in bijective correspondence with  $\Sigma$ -algebras on the same carrier. These are pairs  $\langle C, \llbracket - \rrbracket \rangle$  where  $\llbracket - \rrbracket$  assigns to each  $(f : A) \in \Sigma$  a function  $\llbracket f \rrbracket : C^A \rightarrow C$  from  $A$ -ary tuples of  $C$  elements to  $C$ . The bijection is given by setting  $\llbracket f \rrbracket \langle \xi_a \rangle_{a \in A}$  to be  $c(f \langle \xi_a \rangle_{a \in A})$ .

EFF’s denotational semantics is given by extending MAM’s semantics as follows. Given a type variable assignment  $\theta$ , we assign to each

- ... – handler type: a pair  $\llbracket \Theta \vdash_k X : \mathbf{Hndlr} \rrbracket = \langle C, f \rangle$  consisting of an algebra  $C$  and a function  $f$  into the  $|C|$  carrier of this algebra.

Fig. 11(a) presents how EFF extends MAM’s denotational semantics for types. Each effect  $E$  gives rise to a signature whose operation symbols are the operation names in  $E$  tagged by an element of the denotation of the corresponding parameter type. This signature gives rise to the monad  $E$  denotes. When  $E = \emptyset$ , the induced signature is empty, and gives rise to the identity monad, and so this semantic function extends MAM’s semantics. Handlers handling  $E$ -computations returning  $A$ -values using  $E'$ -computations of type  $C$  denote a pair. Its first component is an  $\llbracket E \rrbracket_\theta$ -algebra structure over the carrier  $|\llbracket C \rrbracket_\theta|$ , which may have nothing to do with the  $\llbracket E' \rrbracket_\theta$ -algebra structure  $\llbracket C \rrbracket_\theta$  already possesses. The second component is a function from  $\llbracket A \rrbracket_\theta$  to the carrier  $|\llbracket C \rrbracket_\theta|$ .

Fig. 11(b) presents how EFF extends MAM’s denotational semantics for terms. The denotation of an operation call  $\text{op}$  makes use of the fact that the effect annotation  $E$  contains the operation name  $\text{op}$ . Consequently, the resulting signature contains an operation symbol  $\text{op}_q$  for every  $q \in \llbracket A \rrbracket_\theta$ . The denotation of  $\text{op}$  is

then the term  $\text{op}_q \langle a \rangle_{a \in [B]_\theta}$ . The denotation of the handling construct uses the Kleisli extension of the second component in the denotation of the handler. The denotation of a handler term defines the  $T_\Sigma$ -algebras by defining a  $\Sigma$ -algebra for the associated signature  $\Sigma$ . The operation clause for  $\text{op}$  allows us to interpret each of the operation symbols associated to  $\text{op}$ . The denotation of the return clause gives the second component of the handler.

|  |  |
|--|--|
| <b>Effects</b>                           | $\llbracket E \rrbracket_\theta := T_{\{\text{op}_p : [A]_\theta \mid (\text{op} : A \rightarrow B) \in E, p \in [A]_\theta\}}$  |
| <b>Handler types</b>                     | $\llbracket A \xRightarrow{E} C \rrbracket := \{\llbracket E \rrbracket\text{-algebras with carrier } \llbracket C \rrbracket\} \times \llbracket C \rrbracket^{[A]}$  |
| (a) Type denotation extensions to Fig. 6 |  |
| <b>Computation terms</b>                 | $\dots \llbracket \text{op } V \rrbracket_\theta (\gamma) := \text{op}_{[V]_\theta \gamma} \langle \text{return } a \rangle_{a \in [B]_\theta}$  |
|  | $\llbracket \text{handle } M \text{ with } H \rrbracket_\theta (\gamma) := \llbracket M \rrbracket_\theta (\gamma) \gg f \text{ where } \llbracket H \rrbracket (\gamma) = \langle D, f : [A] \rightarrow \llbracket C \rrbracket \rangle$ |
| <b>Handler terms</b>                     | $\llbracket \{\text{return } x \mapsto M\} \uplus \{\text{op } p \ k \mapsto N_{\text{op}}\}_{\text{op}} \rrbracket_\theta (\gamma) := \langle D, f \rangle$   |
|  | with $D$ 's algebra structure and $f$ given by:  |
|  | $\llbracket \text{op}_q \rrbracket_D \langle \xi_a \rangle_a := \llbracket N_{\text{op}} \rrbracket_\theta (\gamma[q/p, \langle \xi_a \rangle_a/k] \quad f(a) := \llbracket M \rrbracket_\theta (\gamma[a/x])$                             |
| (b) Term denotation extensions to Fig. 7 |  |

Fig. 11: EFF denotational semantics

We use the lifting in Kammar's [28] thesis to prove adequacy:

**Theorem 3 (adequacy).** *Denotational equivalence implies contextual equivalence: for all  $\Gamma \vdash_E P, Q : X$ , if  $\llbracket P \rrbracket = \llbracket Q \rrbracket$  then  $P \simeq Q$ .*

As a consequence, we obtain a new proof for Kammar et al.'s [30] strong normalisation — well-typed programs that handle all their effects return a value:

**Corollary 4 (soundness and strong normalisation).** *All well-typed, effect-free closed ground returners must reduce to a normal form: for all  $\vdash_\emptyset M : FG$  there exists some  $\vdash V : G$  such that  $\llbracket \text{return } V \rrbracket = \llbracket M \rrbracket$  and  $M \rightsquigarrow^* \text{return } V$ .*

Our Abella formalisation includes additional progress and preservation theorems.

## 4 Monadic reflection: MON

Languages that use monads as an abstraction for user-defined effects employ other mechanisms to support them, usually an overloading resolution mechanism, such as type-classes in Haskell and Coq, and functors/implicits in OCaml. As a consequence, such accounts for monads do not study them as an abstraction in their own right, and are intertwined with implementation details and concepts stemming from the added mechanism. Filinski's [17–20] work on monadic reflection serves precisely this purpose: a calculus in which user-defined monads stand independently.

|  |  |  |              |
|--|--|--|--------------|
| $T ::=$  | monads   | $M, N ::= \dots$   | computations |
| <b>where</b> { <b>return</b> $x = M$ ;         | return clause  | $\hat{\mu}(N)$   | reflect      |
| $y \gg= f = N$ }                               | bind clause  | $[N]^T$  | reify        |
| (a) Syntax extensions to Fig. 2                |  |  |              |
| <hr/>  |  |  |              |
| <b>Frames and contexts</b>                     | $\dots \mathcal{F} ::= \mathcal{B} \mid [ ]^T$   | computation frames   | $\dots$      |
| <b>Beta reduction</b>                          | $\dots$  | for every $T = \mathbf{where} \{ \lambda x. N_u; \lambda y. \lambda f. N_b \}$ : |              |
| ( <i>ret</i> )                                 | $[ \mathbf{return} V ]^T \rightsquigarrow_{\beta} N_u[V/x]$  |  |              |
| ( <i>reflection</i> )                          | $[ \mathcal{H}[\hat{\mu}(N)] ]^T \rightsquigarrow_{\beta} N_b[\{N\}/y, \{(\lambda x. [\mathcal{H}[\mathbf{return} x]]^T)\}/f]$ |  |              |
| (b) Operational semantics extensions to Fig. 3 |  |  |              |

Fig. 12: MON

Fig. 12(a) presents MON's syntax. The **where** {**return**  $x = N_u$ ;  $y \gg= f = N_b$ } construct binds  $x$  in the term  $N_u$  and  $y$  and  $f$  in  $N_b$ . The term  $N_u$  describes the unit and the term  $N_b$  describes the Kleisli extension/bind operation. We elaborate on the choice of the keyword **where** when we describe MON's type system. Using monads, the programmer can write programs as if the new effect was native to the language. We call the mode of programming when the effect appears native the *opaque* view of the effect. In contrast, the *transparent* mode occurs when the code can access the implementation of the effect directly in terms of its defined monad. The *reflect* construct  $\hat{\mu}(N)$  allows the programmer to graft code executing in transparent mode into a block of code executing in opaque mode. The *reify* construct  $[N]^T$  turns a block of opaque code into the result obtained by the implementation of the effect.

Fig. 12(b) describes the extension to the operational semantics. The *ret* transition uses the user-defined monadic return to reify a value. To explain the *reflection* transition, note that the hoisting context  $\mathcal{H}$  captures the continuation at the point of reflection, with an opaque view of the effect  $T$ . The reflected computation  $N$  views this effect transparently. By reifying  $\mathcal{H}$ , we can use the user-defined monadic bind to graft the two together.

Fig. 13 presents the natural extension to MAM's kind and type system for monadic reflection. Effects are a stack of monads. The empty effect is the identity monad. A monad  $T = \mathbf{where} \{ \mathbf{return} x = M; y \gg= f = N \}$  can be *layered* on top of an existing stack  $E$ :

$$E \prec \mathbf{instance\ monad} (\alpha.C) \mathbf{where} \{ \mathbf{return} x = M; y \gg= f = N \}$$

The intention is that the type constructor  $C[-/\alpha]$  has an associated monad structure given by the bodies of the return  $M$  and the bind  $N$ , and can use effects from the rest of the stack  $E$ . To be well-kinded,  $C$  must be an  $E$ -computation, and  $T$  needs to be a well-typed monad, i.e., the return should have type  $C[A/\alpha]$  when substituted for some value  $V : A$ , and the bind should implement a Kleisli extension operation.

The choice of keywords for monads and their types is modelled on their syntax in Haskell. We stress that our calculus does not, however, include a type-class



$$\begin{array}{c}
E ::= \dots \mid E \prec \mathbf{instance\ monad}(\alpha.C)T \quad \text{layered monad} \\
\\
\text{(a) Kinds and types extension to Fig. 4} \\
\hline
\text{Effect kinding} \quad \dots \quad \frac{\Theta, \alpha \vdash_k C : \mathbf{Comp}_E \quad \vdash_m T : E \prec \mathbf{instance\ monad}(\alpha.C)T}{\Theta \vdash_k E \prec \mathbf{instance\ monad}(\alpha.C)T : \mathbf{Eff}} \\
\text{Monad typing} \quad \boxed{\Theta \vdash_m T : E} \\
\\
\frac{\Theta, \alpha; x : \alpha \vdash_E N_u : C \quad \Theta, \alpha, \beta; y : U_E C, f : U_E(\alpha \rightarrow C[\beta/\alpha]) \vdash_E N_b : C[\beta/\alpha]}{\Theta \vdash_m \mathbf{where} \{ \mathbf{return} \ x = N_u; y \gg f = N_b \} :} \\
E \prec \mathbf{instance\ monad}(\alpha.C) \mathbf{where} \{ \mathbf{return} \ x = N_u; y \gg f = N_b \} \\
\\
\text{Computation typing} \quad \dots \quad \frac{\Theta; \Gamma \vdash_E N : C[A/\alpha]}{\Theta; \Gamma \vdash_{E \prec \mathbf{instance\ monad}(\alpha.C)T} \hat{\mu}(N) : FA} \\
\frac{\Theta \vdash_m T : E \prec \mathbf{instance\ monad}(\alpha.C)T \quad \Theta; \Gamma \vdash_{E \prec \mathbf{instance\ monad}(\alpha.C)T} N : FA}{\Theta; \Gamma \vdash_E [N]^T : C[A/\alpha]} \\
\text{(b) Typing extensions to Fig. 5}
\end{array}$$

Fig. 13: MON kind and type system

mechanism. The *type* of a monad contains the return and bind *terms*, which means that we need to check for equality of terms during type-checking, for example, to ensure that we are sequencing two computations with compatible effect annotation. For our purposes,  $\alpha$ -equivalence suffices. This need comes from our choice to use structural, anonymous, monads. In practice, monads are given *nominally*, and two monads are compatible if they have exactly the same name. It is for this reason also that the bodies of the return and the bind operations must be closed, apart from their immediate arguments. If they were allowed to contain open terms, types in type contexts would contain these open terms through the effect annotations in thunks, requiring us to support dependently-typed contexts. The monad abstraction is parametric, so naturally requires the use of type variables, and for this reason we include type variables in the base calculus MAM. We choose monads to be structural and closed to keep them closer to the other abstractions and to reduce the additional lingual constructs involved.

Our calculus deviates from Filinski's [20] in the following ways. First, our effect definitions are local, whereas Filinski's allows nominal declaration of new effects only at the top level. Because we do not allow the bodies of the return of the bind to contain open terms, this distinction between the two calculi is minor. As a consequence, effect definitions in both calculi are *static*, and the monadic bindings can be resolved at compile time. Filinski's calculus also includes a sophisticated *effect-basing* mechanism, that allows a computation to immediately use, via reflection, effects from any layer in the hierarchy below it, whereas our calculus only allows reflecting effects from the layer immediately below. In the presence of Filinski's type system, this deviation does not significantly change the

expressiveness of the calculus: the monad stack is statically known, and, having access to the type information, we can insert multiple reflection operations and lift effects from lower levels into the current level.

As with EFF, MON's ground types are the same as MAM's, and we omit the full definition of program contexts. While we can define an observational equivalence relation in the same way as for MAM and EFF, we will not do so. Monads as a programming abstraction have a well-known conceptual complication — user-defined monads must obey the *monad laws*. These laws are a syntactic counterpart to the three equations in the definition of (set-theoretic/categorical) monads. The difficulty involves deciding what equality between such terms means. The natural candidate is observational equivalence, but as the contexts can themselves define additional monads, it is not straightforward to do so. Giving an acceptable operational interpretation to the monad laws is an open problem in this area. We avoid it by giving a *partial* denotational semantics to MON.

**Effects**  $\dots \llbracket E \prec \text{instance monad } (\alpha.C) N_u N_b \rrbracket_\theta := \langle T, \text{return}, \gg \rangle$  where

$$\begin{aligned} TX &:= \left\lfloor \llbracket C \rrbracket_{(\theta[\alpha \mapsto X])} \right\rfloor & \text{return}^X &:= \llbracket N_u \rrbracket_{(\theta[\alpha \mapsto X])} : X \rightarrow TX \\ \gg^{X,Y} &:= \llbracket N_b \rrbracket_{(\theta[\alpha_1 \mapsto X, \alpha_2 \mapsto Y])} : TX \rightarrow (X \rightarrow TY) \rightarrow TY. \end{aligned}$$

provided these form a monad.

(a) Type denotation extensions to Fig. 6

**Monads**  $\llbracket \Theta \vdash_m T : E \rrbracket := \llbracket E \rrbracket$

**Computation terms**  $\dots \llbracket [N]^T \rrbracket(\gamma) := \llbracket N \rrbracket(\gamma) \quad \llbracket \hat{\mu}(N) \rrbracket(\gamma) := \llbracket N \rrbracket(\gamma)$

(b) Term denotation extensions to Fig. 7

Fig. 14: MON denotational semantics

We extend MAM's denotational semantics to MON as follows. Given a type variable assignment  $\theta$ , we assign to each

- $\dots$  – monad type and effect: a monad  $\llbracket \Theta \vdash_m T : E \rrbracket \theta = \llbracket \Theta \vdash_k E : \mathbf{Eff} \rrbracket \theta$ , if the sub-derivations have well-defined denotations, and this data does indeed form a set-theoretic monad.

Consequently, the denotation of any derivation is undefined if at least one of its sub-derivations has undefined semantics. Moreover, the definition of kinding judgement denotations now depend on term denotation.

Fig. 14(a) shows how MON extends MAM's denotational semantics for types. The assigned type-constructor, and user-defined return and bind, if well-defined, have the appropriate type to give the structure of a monad, and the semantics's definition posits they do. To appreciate the extension to the term semantics from Fig. 14(b), recall that:  $T_{\llbracket E \prec \text{instance monad } (\alpha.C) T \rrbracket} X = \left\lfloor \llbracket C \rrbracket_{(\theta[\alpha \mapsto X])} \right\rfloor$  and

therefore, semantically, we can view any computation of type and kind:

$$\Theta \vdash_k FA : \mathbf{Comp}_{E \prec \text{instance monad}(\alpha.C)T}$$

as an  $E$ -computation of type  $C[A/\alpha]$ .

We define a *proper derivation* to be a derivation whose semantics is well-defined for all type variable assignments, and a *proper term or type* to be a term or type that has a proper derivation. Like the previous cases, we can now extend the denotational semantics to give a partial semantics to program context derivations. We define *proper contexts*, and, unlike the previous cases, the definition of observational equivalence  $\simeq$  is only applicable to proper terms and restricted to proper contexts.

**Theorem 5 (adequacy).** *Denotational equivalence implies contextual equivalence: for all proper derivations  $\Theta; \Gamma \vdash_E P, Q : X$ , if  $\llbracket P \rrbracket = \llbracket Q \rrbracket$  then  $P \simeq Q$ .*

Consequently:

**Corollary 6 (soundness and strong normalisation).** *All well-typed closed ground returners must reduce to a unique normal form: for all  $; \vdash_\emptyset M : FG$  there exists some  $; \vdash V : G$  such that  $\llbracket \mathbf{return} V \rrbracket = \llbracket M \rrbracket$  and  $M \rightsquigarrow^* \mathbf{return} V$  and, moreover, all intermediate steps are proper terms.*

Unlike for MAM and EFF, as this corollary does not apply to improper terms, our Abella formalisation of progress and preservation theorems substantially improves on this result.

In contrast to EFF the semantics for MON is finite:

**Lemma 7 (Finite denotation property).** *For every type variable assignment  $\theta = \langle X_\alpha \rangle_{\alpha \in \Theta}$  of finite sets, every proper types  $A$  and  $C$  denote finite sets.*

## 5 Delimited control: DEL

Delimited control operators can implement algorithms with sophisticated control structure, such as tree-fringe comparison, and other control mechanisms, such as coroutines [13] yet enjoy an improved meta-theory in comparison to their undelimited counterparts [16]. The operator closest in spirit to handlers,  $\mathbf{S_0}$  pronounced “shift zero”, was introduced by Danvy and Filinski [9] as part of a systematic study continuation-passing-style conversion.

Fig. 15(a) presents the extension DEL. The construct  $\mathbf{S_0}k.M$  captures the current continuation and binds it to  $k$ , and replaces it with  $M$ . The construct  $\langle M|x.N \rangle$ , which we will call “reset”, delimits any continuations captured by shift inside  $M$ . Once  $M$  runs its course and returns a value, this value is bound to  $x$  and  $N$  executes. For delimited control cognoscenti this construct is known as “dollar”, and it is capable of macro expressing the entire CPS hierarchy [46].

The extension to the operational semantics in Fig. 15(b) reflects this description. The *ret* rule states that once the delimited computation returns a value,

$$M, N ::= \dots \mid \mathbf{S_0}k.M \quad \text{shift-0} \mid \langle M|x.N \rangle \quad \text{reset}$$

(a) Syntax extensions to Fig. 2

---


$$\begin{array}{ll} \textbf{Frames and contexts} & \dots \quad \mathcal{F} ::= \dots \mid \langle [\ ]|x.N \rangle \text{ computation frame} \\ \textbf{Beta reduction} & \begin{array}{l} (\textit{ret}) \quad \langle (\mathbf{return} V)|x.M \rangle \rightsquigarrow_{\beta} M[V/x] \\ (\textit{capture}) \quad \langle \mathcal{H}[\mathbf{S_0}k.M]|x.N \rangle \rightsquigarrow_{\beta} M[\lambda y. \langle \mathcal{H}[\mathbf{return} y]|x.N \rangle / k] \end{array} \end{array}$$

(b) Operational semantics extensions to Fig. 3

Fig. 15: DEL

this value is substituted in the remainder of the reset computation. For the *capture* rule, the definition of hoisting contexts guarantees that in the reduct  $\langle \mathcal{H}[\mathbf{S_0}k.M]|x.N \rangle$  there are no intervening resets in  $\mathcal{H}$ , and as a consequence  $\mathcal{H}$  is the delimited continuation of the evaluated shift. After the reduction takes place, the continuation is re-wrapped with the reset, while the body of the shift has access to the enclosing continuation. If we were to, instead, not re-wrap the continuation with a reset, we would obtain the control/prompt-zero operators, cf. Shan’s [57] analysis of macro expressivity relationships between these two, and other, variations on delimited control.

$$E ::= \dots \mid E, C \quad \text{enclosing continuation type}$$

(a) Kinds and types extensions to Fig. 4

---


$$\begin{array}{ll} \textbf{Effect kinding} & \dots \quad \frac{\Theta \vdash_k E : \mathbf{Eff} \quad \Theta \vdash_k C : \mathbf{Comp}_E}{\Theta \vdash_k E, C : \mathbf{Eff}} \\ \textbf{Computation typing} & \dots \quad \frac{\Theta; \Gamma, k : U_E(A \rightarrow C) \vdash_E M : C}{\Theta; \Gamma \vdash_{E,C} \mathbf{S_0}k.M : FA} \quad \frac{\Theta; \Gamma \vdash_{E,C} M : FA \quad \Theta; \Gamma, x : A \vdash_E N : C}{\Theta; \Gamma \vdash_E \langle M|x.N \rangle : C} \end{array}$$

(b) Kinding and typing extensions to Fig. 5

Fig. 16: DEL kind and type system

Fig. 16 presents the natural extension to MAM’s kind and type system for delimited control. It is based on Danvy and Filinski’s [8] description, who were the first to propose a type system for delimited control. Effects are now a stack of computation types, with the empty effect standing for the empty stack. The top of this stack is the return type of the currently delimited continuation. Thus, as Fig. 16(b) presents, a shift pops the top-most type off this stack and uses it to type the current continuation, and a reset pushes the return value of the delimited returner onto it.

In this type system, the return type of the continuation remains fixed inside every reset. Ongoing work on type systems for delimited control (see [34] for

a substantial list of references) focuses on type systems that allow *answer type modification*, as these can express typed printf and type-state computation (see, e.g., Asai’s analysis [1]).

Our Abella formalisation establishes:

**Theorem 8 (Safety).** *Well-typed programs don’t go wrong: for all closed, ground returners  $\Theta; \vdash_\emptyset M : FG$ , either  $M \rightsquigarrow N$  for some  $\Theta; \vdash_\emptyset N : FG$  or else  $M = \text{return } V$  for some  $\Theta; \vdash V : G$ .*

Using the translation from DEL to MON we present in the next section, DEL inherits the strong normalisation property from MON.

## 6 Macro translations

Felleisen [14] argues that the usual notions of computability and complexity reduction do not capture the expressiveness of general-purpose programming languages. The Church-Turing thesis and its extensions assert that any reasonably expressive model of computation can be efficiently reduced to any other reasonably expressive model of computation. As many standard algorithms used in compilation and interpretation, such as Hindley-Milner type inference [31, 45], already have worst-case exponential time complexity, we can reduce every general-purpose calculus into another without changing the asymptotic complexity of compilation or interpretation. As an alternative, Felleisen introduces *macro translation*: a *local* reduction of a language extension, in the sense that it is homomorphic with respect to the syntactic constructs, and *conservative*, in the sense that it does not change the core language. We extend this concept to local translations between conservative extensions of a shared core.

Out of the six possible macro-translations, the ideas behind the following four already appear in the literature:  $\text{DEL} \rightarrow \text{MON}$  [61],  $\text{MON} \rightarrow \text{DEL}$  [17],  $\text{DEL} \rightarrow \text{EFF}$  [5], and  $\text{EFF} \rightarrow \text{MON}$  [30]. We use  $\text{DEL} \rightarrow \text{MON}$  and its meta-theoretic properties in the sequel and so recall it in prose alongside the full details of the new translations  $\text{MON} \rightarrow \text{EFF}$  and  $\text{EFF} \rightarrow \text{DEL}$ . The Abella formalisation contains the full details of the six translations and their correctness proofs.

Three translations formally simulate the source calculus by the target calculus:  $\text{MON} \rightarrow \text{DEL}$ ,  $\text{DEL} \rightarrow \text{EFF}$ , and  $\text{MON} \rightarrow \text{EFF}$ . The other translations,  $\text{EFF} \rightarrow \text{MON}$ ,  $\text{EFF} \rightarrow \text{DEL}$ , and  $\text{DEL} \rightarrow \text{MON}$  introduce suspended redexes that invalidate simulation on the nose. These are analogous to *administrative redexes* in continuation-passing-style (CPS) transformations, and may be eliminated by more sophisticated translations. Danvy et al. [10] give a general survey of compact CPS transformations. To keep our translations simple, we adopt a relaxed variant of simulation: for each reduction relation  $\rightsquigarrow$ , let  $\rightsquigarrow_{\text{cong}}$  be the smallest relation containing  $\rightsquigarrow$  that is closed under the term formation constructs. We say that a translation  $M \mapsto \underline{M}$  is a simulation *up to congruence* if for every reduction  $M \rightsquigarrow N$  in the source calculus we have  $\underline{M} \rightsquigarrow_{\text{cong}}^+ \underline{N}$  in the target calculus.

*Translation notation* We define translations  $S \rightarrow T$  from each source calculus  $S$  to each target calculus  $T$ . We allow translations to be *hygienic* and introduce fresh binding occurrences. We write  $M \mapsto \underline{M}$  for the translation at hand. We include only the non-homomorphic cases in the definition of each translation.

*Delimited continuations as monadic reflection.* We adapt Wadler's [61] analysis of delimited control using monads. Let  $\text{Cont}$  be the continuation monad [47]:

$$\text{Cont} := \mathbf{where} \{ \mathbf{return} \ x = \lambda c.c! \ x; m \ggg f = \lambda c.m! \ \{ \lambda y.f! \ y \ c \} \}$$

**Lemma 9.** *For all  $\Theta \vdash_k E : \mathbf{Eff}$ ,  $\Theta \vdash_k C : \mathbf{Comp}_E$ , the  $\text{Cont}$  monad is proper:*

$$\Theta \vdash_m \text{Cont} : E \prec \mathbf{instance \ monad} (\alpha.U_E(\alpha \rightarrow C) \rightarrow C) \text{Cont}$$

Using  $\text{Cont}$ , define the macro translation  $\text{DEL} \rightarrow \text{MON}$  as follows:

$$\underline{\mathbf{Sok.M}} := \hat{\mu}(\lambda k.\underline{M}) \quad \langle \underline{M|x.N} \rangle := [\underline{M}]^{\text{Cont}} \{ \lambda x.\underline{N} \}$$

**Theorem 10** (**DEL  $\rightarrow$  MON correctness**). *MON simulates DEL up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^+ \underline{N}$$

$\text{DEL} \rightarrow \text{MON}$  extends to a macro translation at the type level:

$$\underline{E}, \underline{C} := \underline{E} \prec \mathbf{instance \ monad} (\alpha.U_E(\alpha \rightarrow \underline{C}) \rightarrow \underline{C}) \text{Cont}$$

**Theorem 11** (**DEL  $\rightarrow$  MON preserves typeability**). *Every well-typed DEL phrase  $\Theta; \Gamma \vdash_E P : X$  translates into a proper well-typed MON phrase:  $\Theta; \underline{\Gamma} \vdash_E \underline{P} : \underline{X}$ .*

We use this result to extend the meta-theory of DEL:

**Corollary 12** (**DEL's strong normalisation**). *All well-typed closed ground returners in DEL must reduce to a unique normal form: if  $; \vdash_\emptyset M : FG$  then there exists  $V$  such that  $; \vdash V : G$  and  $M \rightsquigarrow^* \mathbf{return} \ V$ .*

*Monadic reflection as effect handlers.* We simulate reflection as an operation and reification as a handler. Formally, for every anonymous monad  $T$  given by  $\mathbf{where} \{ \mathbf{return} \ x = N_u; y \ggg f = N_b \}$  we define  $\text{MON} \rightarrow \text{EFF}$  as follows:

$$\underline{T} := \{ \mathbf{return} \ x \mapsto \underline{N_u} \} \uplus \{ \mathbf{reflect} \ y \ f \mapsto \underline{N_b} \} \quad \begin{aligned} [\underline{M}]^T &:= \mathbf{handle} \ \underline{M} \ \mathbf{with} \ \underline{T} \\ \hat{\mu}(\underline{N}) &:= \mathbf{reflect} \ \{ \underline{N} \} \end{aligned}$$

**Theorem 13** (**MON  $\rightarrow$  EFF correctness**). *EFF simulates MON on the nose:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^+ \underline{N}$$

$\text{MON} \rightarrow \text{EFF}$  does not preserve typeability. Define the environment/reader monad:

$$\begin{aligned} \text{Reader} &:= \mathbf{where} \{ \mathbf{return} \ x = \lambda e.\mathbf{return} \ x; m \ggg f = \lambda e.x \leftarrow m! \ e; f! \ x \} \\ \vdash_m \text{Reader} : \emptyset \prec \mathbf{instance \ monad} (\alpha.\mathbf{bit} \times U_\emptyset(\mathbf{bit} \rightarrow F \mathbf{bit}) \rightarrow F \alpha) \text{Reader} \end{aligned}$$

Then the following proper computation is a ground return of type  $F \mathbf{bit}$  in MON

$$\begin{aligned} & [b \leftarrow \hat{\mu}(\{\lambda e. \mathbf{case} \ e \ \mathbf{of} \ (b, f) \rightarrow \mathbf{return} \ b\}); \\ & \quad f \leftarrow \hat{\mu}(\{\lambda e. \mathbf{case} \ e \ \mathbf{of} \ (b, f) \rightarrow \mathbf{return} \ f\}); \\ & \quad f! \ b]^\text{Reader} \ (\mathbf{inj}_{\text{true}} \ (), \{\lambda b. \mathbf{return} \ b\}) \end{aligned}$$

but its translation into EFF is not typeable: reflection can appear at any type, whereas a single operation is monomorphic. We hypothesise that this observation can be used to prove *no* macro translation  $\text{PROPER MON} \rightarrow \text{TYPEABLE EFF}$  exists.

*Effect handlers as delimited continuations.* Define  $\text{EFF} \rightarrow \text{DEL}$  as follows:

$$\begin{array}{c} \mathbf{handle} \ M \ \mathbf{with} \quad \langle \underline{M} | x. \lambda h. \underline{N}_{\text{ret}} \rangle \\ \quad \{ \mathbf{return} \ x \mapsto \underline{N}_{\text{ret}} \} \quad \{ \lambda y. \mathbf{case} \ y \ \mathbf{of} \ \{ \\ \quad \uplus \{ \text{op}_1 \ p_1 \ k_1 \mapsto \underline{N}_1 \} \quad := \quad \mathbf{inj}_{\text{op}_1} \ (p_1, k_1) \rightarrow \underline{N}_1 \\ \quad \uplus \dots \quad \quad \quad \vdots \\ \quad \uplus \{ \text{op}_n \ p_n \ k_n \mapsto \underline{N}_n \} \quad \quad \mathbf{inj}_{\text{op}_n} \ (p_n, k_n) \rightarrow \underline{N}_n \} \} \\ \hline \underline{\text{op}} \ V := \mathbf{S}_0 k. \lambda h. h! (\mathbf{inj}_{\text{op}} \ (\underline{V}, \{ \lambda y. k! \ y \ h \})) \end{array}$$

We simulate handling by an application of a reset. The continuation of the reset contains the return clause. We apply this reset to a dispatcher function that invokes the corresponding operation clause based on the operation encoded in its argument. We simulate operation invocation by capturing the current continuation and passing it to the current dispatcher together with the parameter with the operation encoded.

**Theorem 14 (EFF  $\rightarrow$  DEL correctness).** *DEL simulates EFF up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow_{\text{cong}}^+ \underline{N}$$

The  $\text{EFF} \rightarrow \text{DEL}$  translation is simpler than Kammar et al.'s [30] who use a global higher-order memory cell storing the handler stack.

**Theorem 15.** *The following macro translations do not exist:*

- $\text{TYPEABLE EFF} \rightarrow \text{PROPER MON}$  *satisfying:*  $M \rightsquigarrow N \implies \underline{M} \simeq \underline{N}$ .
- $\text{TYPEABLE EFF} \rightarrow \text{TYPEABLE DEL}$  *satisfying:*  $M \rightsquigarrow N \implies \underline{M} \simeq \underline{N}$ .

Our proof of the first part hinges on the finite denotation property (Lemma 7). Briefly, assume to the contrary that there was such a translation. Consider a single effect operation symbol  $\text{tick} : 1 \rightarrow 1$  and the terms:

$$\text{tick}^0 := \mathbf{return} \ () \quad \text{tick}^{n+1} := \text{tick}(); \text{tick}^n$$

All these terms have the same type, and by the homomorphic property of the hypothesised translation, their translations all have the same type. By the finite denotation property there are two observationally equivalent translations and by virtue of a macro translation the two original terms are observationally equivalent in EFF. But every distinct pair of  $\text{tick}^n$  terms is observationally distinguishable using an appropriate handler. See Forster's thesis [21] for the full details. The second part follows from Theorem 11.

## 7 Conclusion and further work

We have given a uniform family of formal calculi expressing the common abstractions for user-defined effects: effect handlers, monadic reflection, and delimited control together with their natural type-and-effect systems. We have used these calculi to formally analyse the relative expressive power of the abstractions: monadic reflection and delimited control have equivalent expressivity; both are equivalent in expressive power to effect handlers when types are not taken into consideration; and neither abstraction can macro-express effect handlers and preserve typeability. We have formalised the more syntactic aspects of our work in the Abella proof assistant, and have used set-theoretic denotational semantics to establish inexpressivity and strong-normalisation results.

Further work abounds. We would like to extend the natural type systems such that each translation preserves typeability. We hypothesise that adding polymorphic effect types would allow effect handlers to express delimited control, and that recursive types would allow monadic reflection and delimited control to express effect handlers.

We are also interested in analysing *global* translations between these abstractions. In particular, while monadic reflection and delimited control allow reflection/shifts to appear anywhere inside a piece of code, in practice, library designers define a fixed set of primitives using reflection/shifts and only expose those primitives to users. This observation suggests calculi in which each *reify/reset* is accompanied by declarations of this fixed set of primitives. We conjecture that MON and DEL can be simulated on the nose via a global translation into the corresponding restricted calculus, and that the restricted calculi can be macro translated into EFF while preserving typeability. Such two-stage translations would give a deeper reason why so many examples typically used for monadic reflection and delimited control can be directly recast using effect handlers. Other global pre-processing may also eliminate administrative reductions from our translations and establish simulation on the nose.

The type system for delimited control we consider, while natural, is rather restrictive. We hope future extensions that support answer type modification (see, e.g., [1, 36]) can inform the design of more expressive type systems for effect handlers and monadic reflection, perhaps accounting for type-state [2] and session types [33]. In practice, effect systems are often extended with sub-effecting or effect polymorphism [44, 4, 52, 41, 26, 43]. To these we add effect-forwarding [30] and rebasing [20]. It would be interesting to investigate how such features affect expressivity.

We have taken the perspective of a programming language designer deciding which programming abstraction to select for expressing user-defined effects. In contrast, Schrijvers et al. [56] take the perspective of a library designer for a specific programming language, Haskell, and compare the abstractions provided by libraries based on monads with those provided by effect handlers. They argue that both libraries converge on the same interface for user-defined effects via Haskell’s type-class mechanism.



Felleisen [14] treats macro reduction from an extended language to a restricted language abstractly, proving meta-theoretic results about all such reductions. In contrast, we treat concrete macro reductions between different extensions of a base calculus. We leave the abstract treatment of this generalisation to further work.

Relative expressiveness results are subtle, and the potentially negative results that are hard to establish make them a risky line of research. We view denotational models as providing a fruitful method for establishing such inexpressivity results. It would be interesting to connect our work with that of Laird [38, 39, 37], who analyses the macro-expressiveness of a hierarchy of combinations of control operators and exceptions using game semantics, and in particular uses such denotational techniques to show certain combinations cannot macro express other combinations. We would like to apply similar techniques to compare the expressive power of local effects such as ML-style reference cells with effect handlers.

## References

1. Asai, K.: On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22(3), 275–291 (2009)
2. Atkey, R.: Parameterised notions of computation. *J. Funct. Program.* 19(3-4), 335–376 (2009)
3. Barr, M., Wells, C.: *Toposes, triples, and theories*. Grundlehren der mathematischen Wissenschaften, Springer-Verlag (1985)
4. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. *Logical Methods in Computer Science* 10(4) (2014)
5. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84(1), 108–123 (2015)
6. Bulwahn, L., et al.: *Imperative Functional Programming with Isabelle/HOL*, pp. 134–149. Springer (2008)
7. Danvy, O.: *An Analytical Approach to Programs as Data Objects*. Dsc dissertation, Department of Computer Science, University of Aarhus (2006)
8. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU (1989)
9. Danvy, O., Filinski, A.: Abstracting control. In: *LISP and Functional Programming*, pp. 151–160 (1990)
10. Danvy, O., et al.: On one-pass CPS transformations. *J. Funct. Program.* 17(6), 793–812 (2007)
11. Doczkal, C.: Strong normalization of CBPV. Tech. rep., Saarland University (2007)
12. Doczkal, C., Schwinghammer, J.: Formalizing a strong normalization proof for moggi’s computational metalanguage. In: *LFMTP*, pp. 57–63. ACM (2009)
13. Felleisen, M.: The theory and practice of first-class prompts. In: Ferrante, J., Mager, P. (eds.) *POPL*, pp. 180–190. ACM Press (1988)
14. Felleisen, M.: On the expressive power of programming languages. *Sci. Comput. Program.* 17(1-3), 35–75 (1991)
15. Felleisen, M., Friedman, D.P.: A reduction semantics for imperative higher-order languages, pp. 206–223. Springer (1987)

16. Felleisen, M., et al.: Abstract continuations: A mathematical semantics for handling full jumps. In: LISP and Functional Programming. pp. 52–62 (1988)
17. Filinski, A.: Representing monads. In: POPL. ACM (1994)
18. Filinski, A.: Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 1996)
19. Filinski, A.: Representing layered monads. In: POPL. ACM (1999)
20. Filinski, A.: Monads in action. SIGPLAN Not. 45(1), 483–494 (Jan 2010)
21. Forster, Y.: On the expressive power of effect handlers and monadic reflection. Tech. rep., University of Cambridge (2016)
22. Gacek, A.: The Abella interactive theorem prover (system description). In: Armando, A., et al. (eds.) IJCAR. vol. 5195, pp. 154–161. Springer (2008)
23. Gacek, A.: A Framework for Specifying, Prototyping, and Reasoning about Computational Systems. Ph.D. thesis, University of Minnesota (September 2009)
24. Gordon, A.D. (ed.): POPL 2016, *to appear*. ACM Press (2017)
25. Hermida, C.: Fibrations, logical predicates and related topics. Ph.D. thesis, University of Edinburgh, 1993 (1993)
26. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Chapman, J., Swierstra, W. (eds.) TyDe. pp. 15–27 (September 2016)
27. Hutton, G., Meijer, E.: Monadic parsing in haskell. J. Funct. Program. 8(4), 437–444 (1998)
28. Kammar, O.: An Algebraic Theory of Type-and-Effect Systems. Ph.D. thesis, University of Edinburgh (2014)
29. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimisations. In: POPL. ACM (2012)
30. Kammar, O., et al.: Handlers in action. SIGPLAN Not. 48(9), 145–158 (Sep 2013)
31. Kanellakis, P.C., Mitchell, J.C.: Polymorphic unification and ML typing. In: POPL. pp. 105–115. ACM Press (1989)
32. Katsumata, S.: Parametric effect monads and semantics of effect systems. SIGPLAN Not. 49(1), 633–645 (Jan 2014)
33. Kiselyov, O.: Parameterized extensible effects and session types (extended abstract). In: Chapman, J., Swierstra, W. (eds.) TyDe. pp. 41–42 (2016)
34. Kiselyov, O., Shan, C.: A substructural type system for delimited continuations. In: TLCA. pp. 223–239 (2007)
35. Kiselyov, O., et al.: Extensible effects: an alternative to monad transformers. In: Haskell. pp. 59–70. ACM (2013)
36. Kobori, I., Kameyama, Y., Kiselyov, O.: Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In: WoC 2015. EPTCS, vol. 212, pp. 36–52 (2015)
37. Laird, J.: Combining control effects and their models. Annals of Pure and Applied Logic (2016), *to appear*
38. Laird, J.: Exceptions, continuations and macro-expressiveness. In: ESOP. pp. 133–146 (2002)
39. Laird, J.: Combining and relating control effects and their semantics. In: COS. pp. 113–129 (2013)
40. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
41. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Gordon [24]
42. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer (2004)

43. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: Gordon [24]
44. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL. pp. 47–57. ACM Press (1988)
45. Mairson, H.G.: Deciding ML typability is complete for deterministic exponential time. In: Allen, F.E. (ed.) POPL. pp. 382–401. ACM Press (1990)
46. Materzok, M., Biernacki, D.: A dynamic interpretation of the CPS hierarchy. In: Jhala, R., Igarashi, A. (eds.) APLAS. LNCS, vol. 7705, pp. 296–311. Springer (2012)
47. Moggi, E.: Computational lambda-calculus and monads. In: (LICS. pp. 14–23. IEEE Computer Society (1989)
48. Plotkin, G., Pretnar, M.: A logic for algebraic effects. In: LICS. pp. 118–129 (2008)
49. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: FoSSaCS. Springer-Verlag (2002)
50. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categ. Structures 11(1), 69–94 (2003)
51. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: ESOP. Springer-Verlag (2009)
52. Pretnar, M.: Inferring algebraic effects. Logical Methods in Computer Science 10(3) (2014)
53. Pretnar, M.: An introduction to algebraic effects and handlers. invited tutorial paper. Electr. Notes Theor. Comput. Sci. 319, 19–35 (2015)
54. Reynolds, J.C.: Theories of Programming Languages. Paperback re-issue, Cambridge University Press (2009)
55. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. Constraints 18(2), 269–305 (2013)
56. Schrijvers, T., et al.: Monad transformers and modular algebraic effects. Tech. rep., University of Leuven (2016)
57. Shan, C.: A static simulation of dynamic delimited control. Higher-Order and Symbolic Computation 20(4), 371–401 (2007)
58. Sinkovics, Á., Porkoláb, Z.: Implementing monads for C++ template metaprograms. Science of Computer Programming 78(9), 1600 – 1621 (2013)
59. Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4), 423–436 (2008)
60. Wadler, P.: Comprehending monads. In: LISP and Functional Programming. pp. 61–78 (1990)
61. Wadler, P.: Monads and composable continuations. Lisp and Symbolic Computation 7(1), 39–56 (1994)
62. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)
63. Ziliani, B., et al.: Mtac: A monad for typed tactic programming in Coq. J. Funct. Program. 25 (2015)